Time-Predictable Memory Arbitration for a Java Chip-Multiprocessor

Christof Pitter
Institute of Computer Engineering
Vienna University of Technology, Austria
cpitter@mail.tuwien.ac.at

ABSTRACT

In this paper, we propose an approach to calculate worst-case execution times (WCET) of tasks running on a homogeneous Java multiprocessor. These processors access a shared main memory. Hence, the tasks running on different CPUs may influence the execution times of each other. Therefore, we implemented a time division multiple access arbiter that divides the memory access time into equal time slots, one time slot for each CPU. This memory arbitration allows calculating upper bounds for the execution time of Java bytecodes depending on the number of CPUs, the size of the time slot, and the memory access time. A WCET analysis tool can utilize these results and generate temporal, upper bounds for application tasks. We further explore how the size of the time slot and the number of CPUs in the system influence the WCET results. Furthermore, a real-world application task is used to compare the analyzed results with measured execution times. This paper describes the timing analysis of a time-predictable Java multiprocessor with shared memory.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.4 [**Programming Languages**]: Processors—*Run-time environments, Java*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Microprocessors and microcomputers*

Keywords

Chip-Multiprocessor, Java, Shared Memory, Worst-case Execution Time

1. INTRODUCTION

Multiprocessors and particularly chip-multiprocessors (CMP) are gaining importance in the embedded market. The CMP technology integrates two or more processing units and a sophisticated communication network into a single integrated circuit. A heterogeneous CMP combines different processing units connected with their memories that are customized to one single part of the application. The homogeneous CMP approach consists of identical CPUs to increase the computation power for general-purpose applications. To save resources and keep the price of the semiconductors low, a single shared main memory is used.

Today, many embedded systems are used for applications where real-time behavior is more important than computation power. Such hard real-time systems must undergo a timing analysis. Therefore, the worst-case execution time (WCET) of each task in the system has to be known. The WCET is the maximum execution time of a piece of code. It is the time a process could eventually need to execute under worst conditions on a given processor. In [20], Wilhelm et al. define the goal of WCET analysis concerning the upper bounds of the execution time:

1. they have to be safe and
2. should be as tight as possible.

Most importantly, the calculated upper timing bounds have to be safe in order to ensure hard real-time behavior of the system. The results must not underestimate the WCET values; otherwise, unpredictable behavior of the system could put the mission at risk with potentially serious consequences. Moreover, the upper bounds should be as tight as possible to keep the overestimation low in order to conserve resources.

There are three different methods to estimate the WCET of a task: static analysis; by measurement; or a hybrid approach combining both methods. The large drawback of the measurement-based method is that these estimates cannot reliably guarantee that the worst-case situation – provoked by a special rare occurring input – is part of the measurement [2]. Furthermore, this kind of analysis is rather complex and time-consuming. Measurement-based analysis can be sufficient for soft real-time systems, but the authors believe that static analysis should be the state-of-the-art of modern hard real-time systems. Using static WCET analysis, the WCET of tasks is analyzed before runtime, independent of any input values. The objective of this analysis is to find the path with the maximum execution time of the program code.

This paper proposes a static WCET analysis of a homogeneous CMP with a shared memory. The CMP system is composed of multiple Java Optimized Processor (JOP) [15, 17] cores and a shared memory. The shared memory is uniformly accessible by the homogeneous processing cores. A system-on-chip (SoC) bus connects the devices of the system.

JOP comes with a static WCET analysis tool, which is described

JTRES'08, September 24-26, 2008, Santa Clara, California, USA

by Schoeberl and Pedersen in [18]. A contribution of this paper is the extension of the WCET tool for timing analysis of the multiprocessor system. The key component for real-time analysis of the CMP is the arbiter that divides the memory access bandwidth into time slots, one for each CPU. Hence, we can analyze the WCET of Java bytecodes depending on the size of the time slot, the number of CPUs in the system, and the memory access time. These execution times are the basis for the WCET analysis of tasks. Our approach is described using a simple example. Additionally, we provide measured data of the execution time of the example. The results are obtained by running the application in hardware. Consequently, we are able to compare the analyzed results with measured execution times. Furthermore, the measured and analyzed execution time results of a real-world application show the reliability of the proposed method.

The rest of the paper is structured as follows. Section 2 outlines related work on WCET analysis. Section 3 presents the CMP system. Additionally, we introduce the approach for time-predictable arbitration of the shared memory access. Section 4 summarizes the static WCET analysis based on JOP. Furthermore, it outlines the WCET analysis of the JopCMP system and gives an example. Section 5 compares the obtained WCET results with measured execution times. Finally, Section 6 concludes the paper and provides guidelines for future work.

2. RELATED WORK

WCET analysis is crucial to timing analysis of hard real-time systems. The task set of a real-time system requires timing validation by schedulability analysis [4, 8]. Hence, the WCET of each task has to be calculated. If and only if these upper bounds of the execution times are known, the schedulability analysis can be performed. Consequently, the result of the analysis shows whether the deadlines of the tasks will be met (guaranteeing that all tasks can be executed by the system) or not.

WCET analysis has been an active and well-established research area for years in the uniprocessor domain. Both Puschner and Burns [12], and Wilhelm et al. [20] give a broad overview of the WCET research. Nevertheless, not all these achievements can be applied to multiprocessor systems. They are based on the assumption that tasks are independent and cannot influence each other. Using modern multiprocessors with shared resources (i.e. a shared memory), tasks influence each other's execution times and cannot be analyzed in isolation.

To the best of our knowledge, only one research group (from university of Linköping) has studied the WCET analysis of multiprocessors [1, 13]. These publications are based on a multiprocessor system-on-chip with a shared communication bus, connecting several CPUs with two different types of memory. Each CPU has a private memory and all the processing units share a common memory for communication. A CPU is equipped with instruction and data caches, which are used to fetch data and instructions from the private memory. During execution, a task can only access private memory and no shared data objects. Hence, all input data must be placed into the private memory before the task can start executing. Consequently, in most cases the execution time of the task can never be influenced by other tasks (compare the simple-task model [5]). However, the communication bus serves as a communication interface between the CPUs and the private memories, and the CPUs and the shared memory. If a cache miss occurs during task execution, data has to be fetched from private memory using

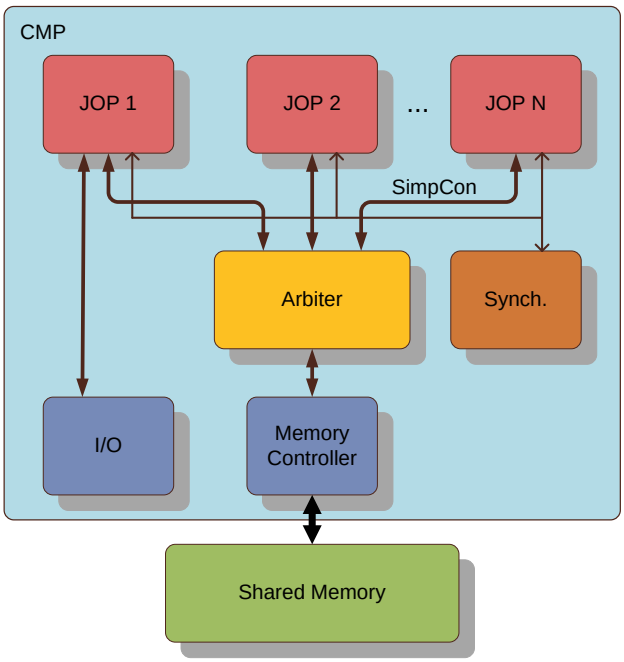

Figure 1: JopCMP Architecture.

the communication bus. Therefore, some sort of bus arbitration is necessary because several CPUs may request a cache line from their private memories simultaneously.

In this paper, we introduce our approach to WCET analysis of a multiprocessor using shared resources. Even though the application tasks running on different CPUs may influence the execution times of each other, we are able to bound the WCET of the real-time tasks.

3. JOPCMP ARCHITECTURE

According to [21], a multiprocessor system consists of three major subsystems: processing elements, memory and an interconnection network. JopCMP implements the symmetric (shared-memory) multiprocessor (SMP) model [3]. Several JOPs provide the basis of the homogeneous CMP as depicted in Figure 1. JOP is an implementation of the Java Virtual Machine (JVM) in hardware [15, 17]. It features a stack cache for the private data of each thread. Additionally, a kind of instruction cache (called method cache) limits the memory access frequency and increases the processing power. These real-time processing elements perform computations in parallel. Instructions and data are stored in a single shared memory. The interconnection network is responsible for connecting multiple processors with the memory. An arbiter is part of this network and controls the memory access to the shared memory. An SoC bus, called SimpCon [16], is used to connect the processing cores to the arbiter, and the arbiter to the shared memory. We consider the synchronization of shared data as a further major subsystem of an SMP. It is responsible for coordinating access to shared objects. A detailed description of the JopCMP architecture can be found in [10].

3.1 Arbitration Challenge

The arbitration of a time-predictable CMP with a shared memory can be divided into two closely coupled challenges:

- Control of the shared memory access
- Timing analysis of the memory access

The arbiter is responsible for partitioning the memory access bandwidth between the CPUs in the system. It controls the memory access of multiple CPUs to the shared memory. Naturally, if one CPU is accessing the shared memory, no other CPU can simultaneously access the memory. They are forced to wait until the CPU has completed its memory transfer. The arbitration unit takes the supervisory and control role for the shared access. Two different arbitration policies exist: dynamic and static arbitration approaches.

A dynamic arbitration policy resolves simultaneous access at runtime. The fixed priority policy is an example. Each CPU of the system is assigned a unique priority. If memory access contention occurs, the CPU with the highest priority will be granted access to the memory. All other CPUs will have to wait. This arbitration algorithm implements a dynamic decision scheme depending on the CPU priorities.

The static arbitration policy strictly defines the access pattern before runtime. Consequently, no arbitration is necessary during execution time. This policy is typical for real-time systems because it provides information for the timing analysis. An example of this policy is the time division multiple access (TDMA) scheme.

What is the problem with time predictability of the CMP? In uniprocessor systems, only one processor accesses the memory and we can predict the WCET of a memory access. However, tasks running on different CPUs influence each others' execution times when they access a shared resource [19], i.e. a shared memory. Therefore, we want to remove the interdependencies between the execution times of tasks. If we already know the pattern of how another task of a CPU accesses the memory, this will make the arbitration and the analysis a lot easier (compare the work of Pitter and Schoeberl in [9]). Usually, we cannot exactly predict the memory access pattern of multiple CPUs. Therefore, we need an arbitration algorithm that is able to bound the WCET of a task running on a CPU, even though tasks executing on other CPUs may also access the main memory. Consequently, the analysis of WCETs is possible.

According to [1, 11, 13], a TDMA based policy guarantees a constant bandwidth to each processor. We agree that this arbitration policy is well suited for time predictability in multiprocessor systems with shared resources. Each processor is assigned a predefined part of the bandwidth. The easiest solution to implement this idea is the division of time. Consequently, each CPU gets an allocated time slot for accessing the shared memory.

We implemented a TDMA arbiter in the hardware description language VHDL. It is fully configurable concerning the number of CPUs and the size of the time slot. Hence, we are capable of running JopCMP with a TDMA arbiter in an FPGA development board. It contains a Cyclone-I FPGA (EP1C12) from Altera and a 1 MB, 15 ns SRAM that provides the main memory of the system. This prototyping hardware is used for experimental measurements of execution times of tasks, as described in Section 5.

4. WCET ANALYSIS

This section starts with a short introduction of the static WCET analysis of JOP. The remaining sections describe the WCET analysis approach of JopCMP.

4.1 Static WCET Analysis based on JOP

Real-time processors like JOP have simpler and less powerful architectures than modern CPUs. Several advanced features that increase the average-case performance (i.e. data caches, out-of-order execution, and branch prediction) are disregarded [17]. Although these methods speedup the execution of programs, they impede the predictability of the timing behavior because the WCET depends on the execution history. Hard real-time processors like JOP benefit from a hardware model that assigns an accurate execution time to each machine instruction.

Using JOP's WCET analysis tool [18], the WCET result of a task can be obtained. A Java program is compiled into class files that include the JVM language instructions called bytecodes. For static WCET analysis, the bytecode sequence is transformed into a directed graph of basic blocks called control flow graph (CFG). Each basic block consists of bytecode instructions. JOP translates each bytecode to a microcode or a sequence of microcode instructions that are executed by the processor. Every microcode has a fixed execution time. Hence, each basic block can be assigned an exact execution time.

Furthermore, flow facts have to be added to the Java program code in advance. In general, this is the only way to bound the loops and calculate the frequency of execution of the basic blocks. The CFG including the flow facts and the mapping to the hardware make the WCET analysis possible using the implicit path enumeration technique (IPET) [6].

4.2 Multiprocessor WCET Approach

With the use of the TDMA arbitration scheme, the WCET of an arbitrary memory access of a CPU can be calculated using Equation 1.

$$WCET_{access} = (n-1) \cdot t_{time\ slot} + t_{access} \quad (1)$$

The unit of $WCET_{access}$ is clock cycles. n specifies the number of CPUs in the system. The more CPUs integrated into the system, the longer the WCET of a single memory access. t_{access} describes the memory access time in clock cycles. The width of a time slot ($t_{time\ slot}$) is given in clock cycles, which is configurable. It should be configured as small as possible to reduce the $WCET_{access}$. The larger the time slot, the higher the $WCET_{access}$ for a single memory access. The minimum size of the time slot is fixed with t_{access}. Otherwise, a processing unit could never successfully access the memory in one time slot.

To find out the overall WCET, the size of the time slot of the arbiter is of major importance. The size of the time slot is dependent on the memory access time. Usually, processors run with a higher clock frequency than memories. The gap between the processor and the memory frequency is further widening [3]. Therefore, the usage of caches is a common approach to reduce the memory access frequency. Nevertheless, the problem with the memory access cannot be circumvented. If the memory access time cannot keep up with the CPU frequency, the processor stalls until the data is available. This delay can be referred to as wait states, which has a large impact on the WCET. A large number for each memory access will degrade the performance of the system.

Type	Bytecode	Memory Area
const	ldc, ldc_w, ldc2_w	Method area
get	getfield, getstatic	Heap
put	putfield, putstatic	Heap
array	aaload, aastore, baload, bastore, caload, castore, daload, dastore, faload, fastore, iaload, iastore, laload, lastore, saload, sastore, arraylength	Heap
call	invokeinterface, invokespecial, invokestatic, invokevirtual	Method area
return	areturn, dreturn, freturn, ireturn, lreturn, return	Method area
new	anewarray, multianewarray, new, newarray	Heap
switch	lookupswitch, tableswitch	Method Area
cast	checkcast, instanceof	Heap

Table 1: Bytecodes accessing the shared memory.

In our approach, we analyze the bytecodes' WCETs that are dependent on:

- Number of JOPs integrated in the CMP
- Size of the time slot
- Memory access time

First, the memory access pattern of each bytecode has to be investigated. The number of JOPs and the size of the time slot have to be defined. This configuration of the system introduces a fixed TDMA memory access scheme where each CPU is assigned a time slot of the TDMA period. Subsequently, all preconditions are accomplished to determine the WCET of each bytecode using the algorithm described in Section 4.4. JOP's WCET analysis tool uses the generated bytecode estimates to calculate the WCET of the Java source code.

4.3 Bytecode Memory Access Pattern

JOP translates most of the bytecodes to its native microcode instructions. Each bytecode is composed of a microcode instruction or a series of microcode instructions. Some bytecodes are actually implemented in hardware. A couple of bytecodes are implemented in Java. The timing analysis of these bytecodes is not part of this work because they have to be analyzed like general Java source code.

According to the JVM specification [7], the heap and the method area are shared data areas, whereas the stack is a private data area for each thread. In JOP, the heap and the method area are located in the main memory. Consequently, all bytecodes that work on these areas have to be carefully examined. Some bytecodes access the memory several times, some only once. Hence, it makes sense to have a closer look at the different instructions. Table 1 summarizes the bytecodes that access the main memory. As stated before, some bytecodes are implemented in Java, i.e. bytecodes of type NEW, SWITCH and CAST. Therefore, they are disregarded in the proposed analysis.

Most memory access patterns of the bytecodes can be statically analyzed; i.e. bytecodes that access the heap and those of type CONST. The pattern is only dependent on the memory access time. If the memory access time is known, the memory access pattern of the bytecodes can be analyzed regardless of the source code of the program. An example of such a bytecode is *ldc*, which pushes a single word constant onto the stack. Therefore, only one memory access to the method area is needed. JOP translates this bytecode into a series of microcodes. If the memory access time is known, the memory access pattern can be specified using JOP's bytecode implementation. Another example is *iaload*, which is implemented in hardware. For the analysis of the memory access pattern, we determine the VHDL implementation in combination with ModelSim simulations.

The memory access patterns of the bytecodes of type CALL and RETURN need a dynamic analysis and are more difficult to attain. Each JOP is equipped with an instruction cache that caches complete Java methods [14]. Consequently, the memory access patterns of these bytecodes vary, depending on the history of the execution. If the method is already in the cache, no additional memory accesses are needed to load the method into the cache. If a cache miss occurs, JOP will have to load the whole method into the cache. Depending on a cache hit or a cache miss and the length of the method that has to be loaded, the access pattern has to be created for each individual occurrence of the bytecode in the source code. Therefore, we integrated the generation of the patterns into the WCET analysis tool where the cache information is available. Again, JOP's microcodes and ModelSim simulations make it possible to analyze the timing of the memory accesses.

Listing 1: Algorithm to find the WCET of the bytecodes.

```
int wcet=0;

for(i=0;i<TDMA_PERIOD;i++){
    execTime=0;
    position=i;

    for(j=0;j<bytecode.length;j++){
        switch(bytecode[j]){
            case NOP:
                execTime++;
                position++;
                break;

            case READ:
                while(tdmaPattern[position]!=1){
                    execTime++;
                    position++;
                }

                execTime++;
                position++;
                break;

            case WRITE:

                ...

                break;
        }
    }
    if(wcet<execTime){
        wcet=execTime;
    }
}
```

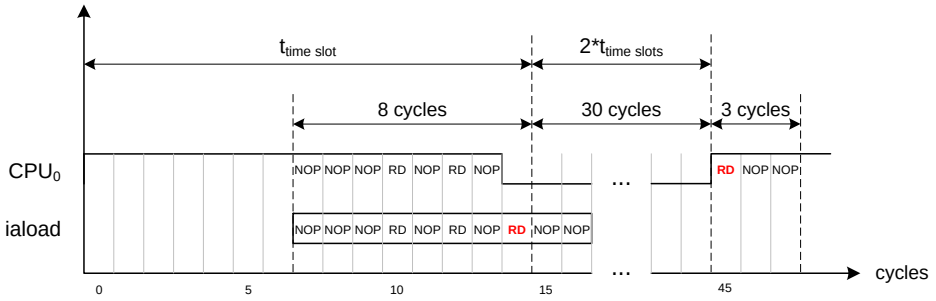

Figure 3: WCET calculation of iaload.

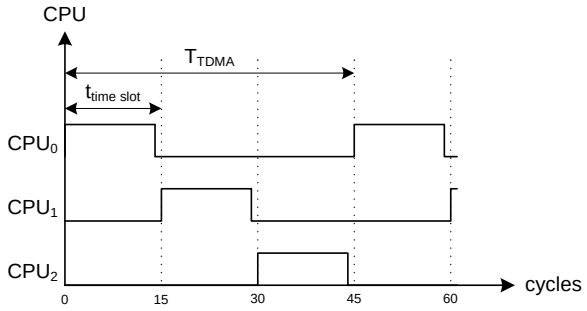

Figure 2: Time slots of the CPUs.

4.4 WCET Analysis of Bytecodes

Listing 1 shows a simplified version of the algorithm to find the WCET of the bytecodes. The inner loop of this algorithm calculates the execution time of the *bytecode* starting at *position*. The *bytecode* describes the memory access pattern of the instruction. It has a predefined length and each element contains either a READ-/WRITE request or a NOP (no memory access). If the indexed element is a NOP, the execution time illustrated by *execTime* will be advanced. Additionally, the variable *position* is increased by 1, which defines the position of the *tdmaPattern* array. If the element of the bytecode equals either a READ or a WRITE access, the *tdmaPattern* will decide whether this CPU will be allowed to access the memory. In case another CPU is on turn to access the memory (the element equals to 0), *execTime* and *position* are advanced until the CPU is allowed access again. The WRITE case of the switch-case statement is similar to the READ case and is therefore omitted from the listing.

The outer loop changes the starting position of the calculation in each iteration. The constant TDMA_PERIOD is defined by the multiplication of the number of CPUs by the size of the time slot. Each iteration of this loop calculates an execution time value of the bytecode. If the new *execTime* is greater than the current worst-case execution time, it will be assigned to *wcet*. Hence, the resulting WCET of the bytecode depending on the number of CPUs and the size of the time slot is available after the last iteration.

WCET Calculation of Bytecode iaload

This simple example exemplifies the calculation of the bytecode *iaload*. JopCMP contains 3 CPUs and the time slot is configured to 15 clock cycles. A read access to the main memory takes 2 cycles. Figure 2 shows the TDMA memory access pattern for each CPU. Time slots to access the shared memory are allocated for each processor. It should be noted that a time slot of 15 cycles permits each CPU to access the memory until the 14^{th} cycle. In the 15^{th} cycle, a read access cannot be permitted. Otherwise, we cannot guarantee that the next CPU is able to access the memory in the first cycle of its time slot. (This originates from the pipelining transactions of the SimpCon specification [16]).

We want to evaluate the WCET of the bytecode *iaload*. The following access pattern is given for *iaload* = {NOP, NOP, NOP, RD, NOP, RD, NOP, RD, NOP, NOP}. We can see that *iaload* performs three read accesses to the memory. The WCET is calculated when the index of the outer loop equals to 7. This scenario is shown in Figure 3. *iaload* starts with an NOP operation in the 8^{th} cycle of the CPU's time slot. We can see that the 3^{rd} RD access of *iaload* cannot immediately be executed. Therefore, it is delayed two time slots until it is allowed to access the memory. The WCET results in 41 cycles.

4.5 Loop Example

In the following, we systematically analyze the WCET of a simple loop to show how the WCET is calculated. Listing 2 shows the source code where a scalar *s* is added to a vector. This loop is parallelizable because each iteration of the statement in the loop body is independent. Therefore, the loop body could be easily executed on different CPUs in parallel.

Listing 2: Simple Loop.

```
for (i=0; i<10; i++) { // @WCA loop=10
    a[i]=a[i]+s;
}
```

As explained before, the WCET estimate for each bytecode accessing the main memory has to be calculated depending on the configuration of the CMP. For this example, JopCMP consists of 3 CPUs and the time slot for each CPU is specified as 15 clock cycles. Consequently, one TDMA period is 45 cycles. The bytecodes and basic blocks of the example, as generated by the WCET analysis tool, are shown in Table 2. The fourth column presents the execution time in clock cycles for each bytecode, and the fifth column gives the execution time for each basic block. If we compare the bytecodes with Table 1, only *iaload* and *iastore* access the main memory. Therefore, their WCETs are dependent on the configuration of the system.

The CFG, illustrated in Figure 4, is constructed from the basic blocks. The vertices represent the basic blocks labeled with their name and execution time. All edges are labeled with an execution

Table 2: Java bytecodes and basic blocks of the loop.

Block	Addr.	Bytecode	Cycles	BB Cycles
B1	0:	iconst_0	1	
	1:	istore_3	1	2
B2	2:	iload_3	1	
	3:	iload_0	1	
	4:	if_icmpge	4	6
B3	7:	aload_1	1	
	8:	iload_3	1	
	9:	aload_1	1	
	10:	iload_3	1	
	11:	iaload	41	
	12:	iload_2	1	
	13:	iadd	1	
	14:	iastore	46	
	15:	iinc	8	
	18:	goto 2	4	105

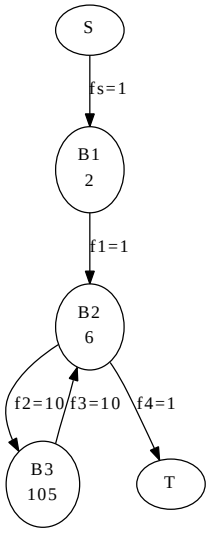

Figure 4: Control flow graph of the simple loop.

frequency. Hence, the WCET can be calculated and the result is 1118 cycles.

We can also measure the execution time of this simple example by running the JopCMP on the FPGA development board described in Section 3. The measured execution time of the loop example results in 987 cycles. This result and the analytical WCET estimate diverge slightly. This overestimation of the analytical result is not surprising because the analysis always takes the WCET for the bytecodes *iaload* and *iastore* into account. In the measurement, some array accesses are executed in fewer clock cycles than the worst case.

5. RESULTS

We use the simple loop as an example to find the WCET estimates for varying system configurations. In Table 3, the analyzed WCETs of different system configurations are shown. The number of CPUs is varied between 1 and 32. Additionally, the size of the time slot changes between 3 and 48 clock cycles. The first row of the results

Table 3: Analyzed WCET of the loop example depending on the system configuration.

Configuration # of CPUs	Time Slot (cycles)	*Analyzed* WCET (cycles)
1	–	488
2	3	708
2	4	668
2	5	728
2	6	708
2	7	728
2	8	768
2	9	698
2	10	718
2	11	738
2	12	758
2	24	998
2	48	1478
4	3	1068
4	6	1068
4	12	1238
4	24	1958
4	48	3398
8	3	1788
8	6	1788
8	12	2198
8	24	3878
8	48	7238
16	3	3228
16	6	3228
16	12	4118
16	24	7718
16	48	14918
32	3	6108
32	6	6108
32	12	7958
32	24	15398
32	48	30278

shows that the WCET of the single JOP is 488 cycles. No number is given for the time slot of this configuration because a single JOP does not have to share the main memory bandwidth.

The second part of the table shows the results of the dual-core JopCMP with varying time slot sizes. In general the WCET increases continuously with larger time slot sizes. The systems with more CPUs show a similar behavior concerning the size of the time slot. When comparing the analyzed WCET results of the 2-way JopCMP, both systems with the time slot equal to 3 and 6 cycles have the same WCET. Only the two bytecodes *iaload* and *iastore* of the loop access the main memory. The execution times of all other bytecodes of the loop are not affected by the size of the time slot. The configuration with the time slot equal to 3 results in a WCET of 22 cycles for *iaload* and 24 cycles for *iastore*. If the time slot is 6 cycles, the WCETs of *iaload*=17 and *iastore*=29 cycles. We can see that the sums of the two WCETs of the bytecodes are the same, independent of whether the size of the time slot equals 3 or 6 cycles. Consequently, the execution times of both loop bodies

Table 4: Analyzed WCET and measured execution time of the loop example.

Configuration # of CPUs	Time Slot (cycles)	Analyzed WCET (cycles)	Measured Exec. (cycles)
1	–	488	488
2	3	708	670
2	6	708	666
2	9	698	554
2	12	758	716
2	24	998	502
2	48	1478	931
3	3	888	838
3	6	888	747
3	9	878	815
3	12	998	735
3	24	1478	765
3	48	2438	1410

are identical. Note that the configuration with a 4-cycle time slot results in the lowest execution time. In this configuration, the sum of *iaload* and *iastore* is minimized.

The more CPUs integrated into the system, the longer the WCETs. The configuration with 4 CPUs and 6-cycle time slot results in a WCET of 1068 cycles. *iaload* executes in 29 cycles and *iastore* in 53 cycles. Even though, the number of CPUs in the system is doubled, the WCET only increases by 51% compared to the 2-way CMP with the same time slot value.

Table 4 compares the measured execution time and the analyzed WCET of the simple loop example. One can see that the measured execution time and the WCET estimate are equal for a single JOP system. This originates from the characteristics of the example. The WCET result and the measured execution time are the same, because only one execution path exists in the code. Furthermore, one can see that the WCET estimates are tight for CMP versions with minor time slots. The configurations with a slot size of 9 cycles result in the lowest WCETs. The measured execution times vary greatly depending on the time slot. It results in 716 cycles for the 2-way CMP with a 12-cycle time slot. The same system with a slot size equal to 24 cycles executes in only 502 cycles. This result shows that *iaload* and *iastore* are frequently executed in only one time slot in the configuration with 24 cycles. In the worst case, they need two time slots to execute, which explains the large difference between the measured and the analyzed execution time.

Furthermore, we used a benchmark called Lift as another example to calculate some WCETs. Lift is a real-world example with an industrial background. This embedded application is a lift controller used in an automation factory. It is part of the embedded Java benchmark suite called JavaBenchEmbedded, as described in [15]. Table 5 shows that the WCET of a single JOP results in 8689 cycles. Each CPU of a dual-core JopCMP with a time slot size of three cycles executes the Lift benchmark in 12391 cycles in the worst case. Therefore, the WCET increases only by 43%. The tri-core CMP version experiences an increase of 83% in the execution time compared to the single JOP. Whereas one JOP executes Lift only once, the CMP configuration executes the benchmark three times in parallel.

Table 5: Analyzed WCET and measured execution time of the Lift benchmark.

Configuration # of CPUs	Time Slot (cycles)	Analyzed WCET (cycles)	Measured Exec. (cycles)	Pessimism (Ratio)
1	–	8689	5830	1.49
2	3	12391	7587	1.63
2	6	12580	7305	1.72
2	9	13621	7623	1.79
2	12	14867	8131	1.83
2	24	19391	8409	2.31
2	48	28618	9499	3.01
3	3	15872	9234	1.72
3	6	16186	8721	1.86
3	9	18072	9720	1.86
3	12	20456	11097	1.84
3	24	29735	12489	2.38
3	48	48106	14211	3.36
4	3	18827	–	–
4	6	19876	–	–
4	9	22621	–	–
4	12	26171	–	–
4	24	40079	–	–
4	48	67594	–	–

The last column of Table 5 illustrates the pessimism of the WCET analysis. Even though we cannot pretend to measure the WCET, the pessimism ratio gives us an idea of the quality of our analyzed results. One can see that there is not much difference between the measurement and the analysis of the single JOP and the 2-way CMP version with a reasonable size of the time slot. Nevertheless, the conservatism of the analysis does not increase greatly with three JOPs. Unfortunately, we are not able to integrate more than 3 JOP cores into the available Cyclone-I FPGA. Therefore, no values of the measured execution times and corresponding pessimism ratios are available for the 4-way CMP. We can see that the pessimism is in an acceptable range for a multiprocessor WCET analysis.

6. CONCLUSION AND FUTURE WORK

In this paper, we have described the extension of JOP's WCET tool for a homogeneous multiprocessor with a shared memory. The key component of real-time analysis of JopCMP is a TDMA arbiter that divides the memory access bandwidth into equal shares. Hence, we can analyze the WCET of Java bytecodes depending on the size of the time slot, number of CPUs in the system and the memory access time. They are used in the WCET analysis tool to calculate WCET estimates. A simple loop example is presented and WCET estimates are compared to real measurement results.

In the future, we will conduct extensive experiments using more benchmarks to investigate the frequencies of the bytecodes that access the main memory. The applications determine the size of the time slot to achieve tight WCET results. It is defined to be short in size if single memory accesses are dominant in the application code. A medium slot size is the solution for frequent field and array accesses and a large time slot size for predominant method cache load accesses. Consequently, we will be able to better jus-

tify, whether to define the size of the time slot larger or smaller. Furthermore, we want to extend the solution and present an idea how tighter WCET estimates can be obtained, in order to increase the accuracy of the analysis.

We will investigate the use of a percentage-based arbitration of the available memory access bandwidth. Hence, the memory bandwidth per CPU will be adjusted, dependent on the workload of the multiple CPUs. For example, if CPU_0 needs 60% of the available memory bandwidth for example, it will receive 60% of the bandwidth share or the time slots accordingly.

Acknowledgement

The research leading to these results has received funding from the Austrian Research Programme FIT-IT under contract number 813039 (TPCM).

7. REFERENCES

[1] A. Andrei, P. Eles, Z. Peng, and J. Rosen. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *VLSI Design*, pages 103–110. IEEE Computer Society, 2008.

[2] A. Ermedahl and J. Engblom. Execution time analysis for embedded real-time systems. pages 35.1 – 35.17. Chapman & Hall/CRC - Taylor and Francis Group, August 2007.

[3] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[4] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *Comput. J*, 29(5):390–395, 1986.

[5] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*. Kluwer Academic Publishers, 1997.

[6] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 88–98, 1995.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[8] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[9] C. Pitter and M. Schoeberl. Time predictable CPU and DMA shared memory access. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, Amsterdam, Netherlands, August 2007.

[10] C. Pitter and M. Schoeberl. Performance Evaluation of a Java Chip-Multiprocessor. In *Proceedings of the IEEE Third Symposium on Industrial Embedded Systems (SIES 2008)*, Montpellier, France, June 2008.

[11] F. Poletti, D. Bertozzi, L. Benini, and A. Bogliolo. Performance Analysis of Arbitration Policies for SoC Communication Architectures. *Design Automation for Embedded Systems*, 8:189–210(22), 200306/09.

[12] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[13] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, pages 49–60. IEEE Computer Society, 2007.

[14] M. Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[15] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[16] M. Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.

[17] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[18] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2006)*, pages 202–211, Paris, France, Oct. 2006. ACM.

[19] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[20] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst*, 7(3), 2008.

[21] W. Wolf. *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.